Calling Toolbox Routines 18

The Macintosh Toolbox is the set of over one thousand routines that give Macintosh applications the consistent interface. This chapter outlines what you have to do to call the routines described in *Inside Macintosh*. It also describes the few differences between the Apple and the THINK C interfaces to the Toolbox.

Contents

Calling Toolbox Routines 351
- Passing arguments to Toolbox routines 352
- Working with Pascal strings 353
- Calling AppleTalk routines 354
- Using MacHeaders 355
- Editing the MacHeaders file 356

The Macintosh Header Files 357
- The Mac #includes folder 358
- The Apple #includes folder 358
- The THINK #includes folder 359
- Working with Pascal routines 362
- Pascal callback routines 362
- Calling Pascal routines indirectly 363

Working with Floating-Point Formats 364
- Working with SANE 364
- The extended type 365
- A note on SANE implementations 366

Calling Toolbox Routines

With THINK C, you can use all of the routines described in *Inside Macintosh I-VI*, including those marked `[Not in ROM]`. To use the Toolbox routines, call them exactly as they appear in *Inside Macintosh*. The only thing you need to know is how to convert the Pascal declarations into C declarations.

Most Macintosh routines are implemented as traps that use Pascal calling conventions. THINK C generates these traps inline instead of generating a subroutine call. For routines marked `[Not In ROM]`, THINK C generates calls to library functions in MacTraps. Calls to these routines also follow Pascal calling conventions.

To use a Macintosh Toolbox routine, you must be using the right library and headers. If you're using `MacHeaders++`, you won't have to include header files for the most common Toolbox routines.

For documentation on HyperCard `XCMD`*s and the Communications Toolbox, contact APDA. See "Apple Programmer's and Developer's Association (APDA)" in Chapter 1, "Welcome."*

This library…	Contains…
MacTraps	Glue for routines in *Inside Macintosh I-V*, definitions for QuickDraw globals, and the Gestalt glue
MacTraps2	Glue for routines in *Inside Macintosh VI* and less common routines
AppleTalk	The alternate AppleTalk library
nAppleTalk	The preferred AppleTalk library
HyperXLib	Glue for HyperCard `XCMD`s
CommToolbox	Library for the Communications Toolbox
SANE	SANE routines
QuickTime	QuickTime glue
OSL	Support for Apple's Object Support Library as described in *Inside Macintosh: Interapplication Communication*

Table 18-1 Macintosh Toolbox libraries

Almost every Macintosh program needs to use MacTraps. If you use routines described in *Inside Macintosh VI*, you should also use MacTraps2.

Passing arguments to Toolbox routines

You don't need to know the details of C and Pascal calling conventions, though you might find the information useful. See "Calling C Routines" and "Calling Pascal Routines" in Chapter 19, "Porting Code."

Since argument declarations in *Inside Macintosh* are given in Pascal, you need to know how to convert them to C. This table gives you the general rule for converting argument declarations from Pascal to C:

If the object is…	Pass…
a `var` parameter	a pointer to the object
4 bytes or smaller	the object
larger than 4 bytes	a pointer to the object

Table 18-2 Rules for converting Pascal declarations

Here are some examples of Pascal declarations and their C counterparts:

Pascal type	C type
`integer`	`short`
`longint`	`long`
`char`	`short`
`Boolean`	`Boolean` or `char`
`Byte`	`Byte` in `struct` declarations, `short` when passed as an argument
`var Byte`	`short *`
`ProcPtr`	See "Working with Pascal routines" later in this chapter.
`Handle`	`Handle`
`var Handle`	`Handle *`
`Ptr`	`Ptr`
`var Ptr`	`Ptr *`

Table 18-3 Examples of converting Pascal declarations

Pascal type	C type
`OSType, ResType`	`OSType, ResType, long`
`packed array [1..4] of char`	`long`
`Str255`	`Str255` or `unsigned char *`
`var Str255`	`Str255` or `unsigned char *`
`StringPtr`	`StringPtr` or `unsigned char *`
`var StringPtr`	`StringPtr *` or `unsigned char **`
`Rect`	`Rect *`
`var Rect`	`Rect *`
`Point`	`Point`
`var Point`	`Point *`
`Extended`	`extended`, or in certain cases, `double`. See "Working with Floating-Point Formats" later in this chapter.

Table 18-3 Examples of converting Pascal declarations *(Continued)*

Working with Pascal strings

Toolbox routines that take strings as arguments expect them to be Pascal strings. Unlike null-terminated C strings, Pascal strings begin with a length byte. To write a Pascal string constant, start the string with `"\P"` or `"\p"`. This is how you would call the QuickDraw routine `DrawString()`:

```
DrawString("\pThis is a Pascal string");
```

Because Pascal strings start with a length byte, the longest a Pascal string can be is 255 bytes. They are *not* null terminated. Pascal string literals are of type `unsigned char[]` by default to agree with the definition of `Str255`. Older versions of THINK C treated Pascal string literals as `char []`. You can change this behavior on the Compiler Settings page of the **THINK C Options** dialog. (For

Symantec C++, Pascal strings are always of type `unsigned char []`).

You can use these routines to convert strings from one form to the other:

To convert a …	Use this function…
C string to a Pascal string	`CtoPstr()`
Pascal string to a C string	`PtoCstr()`

Table 18-4 Functions that convert strings

These routines convert the strings in place and return the converted string. These are their function prototypes:

```
unsigned char *CtoPstr(char *s);
char *PtoCstr(unsigned char *s);
```

Note

If you're not using `MacHeaders`, include `pascal.h` to use these functions.

Calling AppleTalk routines

If your application uses AppleTalk, you should be aware that there are two sets of AppleTalk interfaces. Apple calls the old interface (described in *Inside Macintosh II*) the alternate set. The new interface (described in *Inside Macintosh V)* is called the preferred set. You can use either or both sets of interfaces.

To use the alternate AppleTalk routines, use the library AppleTalk. To use the preferred routines, use nAppleTalk and AppleTalk. In either case, you should include the header file `AppleTalk.h`.

To use this interface…	Use these libraries…
preferred	`nAppleTalk` and `AppleTalk`
alternate	`AppleTalk`

Table 18-5 Choosing an AppleTalk library

Earlier versions of THINK C used two header files, `AppleTalk.h` and `nAppleTalk.h`. This version uses only one header file, `AppleTalk.h`, supplied by Apple. The data structure definitions in the earlier version were adapted from the Pascal definitions in *Inside Macintosh*. The data structure definitions in the current header files

are slightly different, so you may have to make changes to your source code.

Using MacHeaders

THINK C comes with a precompiled header file, `MacHeaders`, which contains the most common declarations you use for writing Macintosh programs. (Symantec C++ comes with `MacHeaders++`.) If you include `MacHeaders` in the project prefix, the THINK Project Manager automatically includes `MacHeaders` in all the files in your project, so you never have to explicitly include common header files like `QuickDraw.h`. (though it doesn't hurt if you do). To edit the project prefix, go to the Prefix page of the **Options** dialog.

Note

The project prefix contains the line `#include <MacHeaders>` by default.

The `MacHeaders` file contains these files:

`Controls.h`	`Desk.h`
`Devices.h`	`Dialogs.h`
`DiskInit.h`	`Errors.h`
`Events.h`	`Files.h`
`Fonts.h`	`Lists.h`
`Memory.h`	`Menus.h`
`Notification.h`	`OSEvents.h`
`OSUtils.h`	`Processes.h`
`Quickdraw.h`	`Resources.h`
`Scrap.h`	`SegLoad.h`
`StandardFile.h`	`TextEdit.h`
`Timer.h`	`ToolUtils.h`
`Types.h`	`Windows.h`
`pascal.h`	`asm.h` (THINK C only)
`LoMem.h` (THINK C only)	`THINK.h` (THINK C only)

These files aren't used as often, so they're not included in `MacHeaders`. You'll have to include them yourself.

`ADSP.h`	`AIFF.h`
`Aliases.h`	`AppleEvents.h`
`AppleTalk.h`	`Balloons.h`
`CommResources.h`	`Connections.h`
`ConnectionTools.h`	`CRMSerialDevices.h`

```
CTBUtilities.h          DatabaseAccess.h
DeskBus.h               Disks.h
Editions.h              ENET.h
EPPC.h                  FileTransers.h
FileTrasferTools.h      Finder.h
FixMath.h               Folders.h
GestaltEqu.h            Graf3D.h
HyperXCmd.h             Icons.h
Language.h              MIDI.h
Packages.h              Palettes.h
Picker.h                PictUtil.h
Power.h                 PPCToolBox.h
Printing.h              PrintTraps.h
QDOffScreen.h           Retrace.h
ROMDefs.h               SANE.h
Script.h                SCSI.h
Serial.h                ShutDown.h
Slots.h                 Sound.h
SoundInput.h            Start.h
SysEqu.h                Terminals.h
TerminalTools.h         Values.h
Video.h                 Traps.h
Components.h            ImageCode.c
ImageCompression.h      MediaHandlers.h
Movies.h                MoviesFormat.h
QuickTimeComponents.h
```

Usually, you'll just use the built-in `MacHeaders`. You can, however, edit the default `MacHeaders` file or make your own precompiled headers.

Editing the MacHeaders file

You might find that in the kinds of programs you write, you frequently refer to a header file that is not already in `MacHeaders`. Or, `MacHeaders` might include some files you never use. You can edit the `MacHeaders` file to suit your needs.

1. Find the file `Mac #includes.c` in the `Mac #includes` folder. (For Symantec C++, note that the file `Mac #includes.cpp` merely includes `Mac #includes.c`.) Duplicate it and give it a new name, such as `My #includes.c`. Open the duplicate with the THINK Project Manager.

2. If you want to change the compile options, edit the `#pragma options` directive. Otherwise, it uses the settings in the **Options** dialog and uses different options, depending on the project that's open.

3. Search for the files you want to add or remove. All `#include` statements are enclosed in conditional compilation directives. To add a file, change the `#if 0` directive to `#if 1`. To remove a file, change the `#if 1` directive to `#if 0`. Some files can't be used together. For more information, see the list below.

4. Choose **Precompile** from the **Source** menu. After the THINK Project Manager precompiles the file, save it as `MacHeaders`. (Precompiled files don't go into the project.) The best place for `MacHeaders` is in the `Mac #includes` folder, but you can save it anywhere in the THINK tree.

The auto-make facility marks the files in the current project for recompilation if you change `MacHeaders`. To let other projects know that `MacHeaders` changed, use the **Make** command from the **Source** menu. Click the Use Disk button to mark the files affected, then click the Make button to recompile them.

When you add or remove files from `MacHeaders`, keep these dependencies in mind:

- You cannot include both `LoMem.h` and `SysEqu.h`.
- If you include both `asm.h` and `Traps.h`, you must include `asm.h` before `Traps.h`. If you include `Traps.h`, do not prefix traps with underscores in inline assembly.

The Macintosh Header Files

THINK C gets its information about the Toolbox routines from the header files in the `Mac #includes` folder. The header files contain C prototypes for all Toolbox routines. In many cases, Toolbox routines are defined as inline functions.

Note

Earlier versions of THINK C had a built-in list of all Toolbox routines and information about the arguments they took. This version does not have such a list. Instead, it uses the information in the header files.

For the most common Toolbox routines, you won't have to include any header files explicitly if you use the `MacHeaders` precompiled header.

The Mac #includes folder

The `Mac #includes` folder contains the following folders and files:

Item	Description
`Apple #includes`	This folder contains all of the Macintosh header files from Apple. Some have been modified for THINK C. The next section describes the modifications.
`Mac #includes.c`	This file is the "source file" for `MacHeaders`, the default precompiled header. Instructions in this file explain how to modify it to create a customized precompiled header.
`MacHeaders`	This is the default precompiled header for THINK C. (Symantec C++ uses `MacHeaders++`.) It contains the symbols from the most commonly used Toolbox manager.
`THINK #includes`	This folder contains header files unique to THINK C. The files are described in more detail in the next section.

Table 18-6 Files in the Mac #includes folder

The Apple #includes folder

This folder contains header files for the Macintosh Toolbox. Most of these files come from Apple, and they are the official interfaces to the Macintosh Toolbox. The following files have been modified for THINK C. Changes are marked in the file with conditional

compilation directives, except for SANE.h, which is completely replaced with a THINK C version.

File	Modification
Connections.h	Change in enum definitions to accommodate different size of int in THINK C.
Files.h	Change type of bit-field to take into account differences in bit-packing.
Packages.h	Add #include <BDC.h> directive. See the description of BDC.h on the following page.
Quickdraw.h	Change in enum definitions to accommodate different size of int in THINK C.
SANE.h	Replace with THINK C version. See the description of SANE.h later in this chapter.
Sound.h	Don't define type Time to avoid conflict with low-memory global named Time.
Types.h	Add special definition of extended that is always 80 bits wide and THINK C's definition of NULL, ((void *) 0).

Table 18-7 Modifications in the Apple #includes folder

If you want to examine the changes, open the header file and search for the symbol THINK_C. (For Symantec C++, search for THINK_CPLUS.)

The THINK #includes folder

This folder contains header files unique to THINK C. These files contain definitions for the inline assembler, routines for working with globals in non-application projects, definitions of low-memory globals, and so on.

Most of these files are included in MacHeaders:

File	Included in MacHeaders?
asm.h	yes (no for Symantec C++)
BDC.h	yes
LoMem.h	yes (no for Symantec C++)
pascal.h	yes
SANE.h	no
SetUpA4.h	no
THINK.h	yes (no for Symantec C++)

Table 18-8 Files in both THINK #includes folder and MacHeaders

asm.h
This header file contains definitions for the inline assembler. It contains definitions for the trap modifier bits and inline function definitions that are used differently in assembly language than in C. To learn about the inline assembler, see Chapter 20, "Writing Inline Assembly."

BDC.h
This header file contains the function prototypes for the routines in the Binary-Decimal Conversion Package, StringToNum() and NumToString(). These routines are declared in the Apple header file, Packages.h.

Because programmers use these functions often, and because Packages.h brings in Script.h, which is rather large, the THINK C version breaks out these functions into a smaller header file. This smaller file can be included in MacHeaders.

LoMem.h
This file contains definitions of the low-memory globals that take advantage of THINK C's absolute address extension.

The file has been modified so the same calling conventions can be used with Symantec C++. In Symantec C++, LoMem.h cannot be precompiled or be part of a precompiled header. To use it, you must explicitly #include it in your prefix.

The corresponding Apple header file is SysEqu.h. LoMem.h does not define each low-memory global, only the most common ones. If you want to use a low-memory global that's not defined in LoMem.h, you can use THINK C's absolute address extension to define it in your source file.

Warning

You can use either LoMem.h or SysEqu.h. You cannot include both header files in the same source file, or you will get compiler errors.

If you use LoMem.h, you can treat low-memory globals the same way you would any other global variable:

```
#include <LoMem.h>   // or MacHeaders
void F(void)
{
   WindowPeek  w;

   w = WindowList;   // Get first window
   ...
}
```

If you use SysEqu.h, you have to cast the value to use it:

```
#include <SysEqu.h>  // but not MacHeaders
void G(void)
{
   WindowPeek  w;

   w = * (WindowPeek *) WindowList;
}
```

If you're programming exclusively in THINK C, you should use LoMem.h. If you want to be able to compile your program with MPW C, use SysEqu.h.

Note that MacHeaders includes LoMem.h by default. If you want to use MacHeaders and SysEqu.h, you should build a new version of MacHeaders that includes SysEqu.h instead of LoMem.h.

pascal.h
This header file contains declarations for the functions CtoPstr() and PtoCstr() that translate C strings into Pascal strings and vice versa. See "Working with Pascal strings" earlier in this chapter for a full description.

SANE.h
This is the header file for the THINK C implementation of SANE, the Standard Apple Numerics Environment. This header file contains definitions of SANE types and declarations of SANE functions. See "Working with SANE" later in this chapter for more information.

SetUpA4.h
This header file contains the utility routines that let you use global variables in desk accessories, device drivers, and code resources. See "Global data in drivers" and "Global data in code resources" in Chapter 13, "Building Projects," for more information.

THINK.h
This header file contains definitions and declarations that have been part of the previous versions of the THINK C environment but that have no counterparts in the Apple header files. For example, the uppercase versions of the Boolean values `TRUE` and `FALSE` are defined here, as are the macros `topLeft` and `botRight` for extracting points from rectangles.

Note

Symantec C++ does not use `THINK.h`.

Working with Pascal routines

Because the Macintosh Toolbox routines expect to be called with Pascal calling conventions, you will probably need to write functions in C that behave as though they were written in Pascal.

Pascal callback routines

Some Macintosh Toolbox routines take a pointer to another function as an argument. Those routines then call the function you passed in. The function you provide is called a callback routine. Toolbox routines expect callback routines to follow Pascal calling conventions. (To learn about the difference between Pascal and C calling conventions, see Chapter 20, "Writing Inline Assembly.")

THINK C gives you a way to write functions that behave as though they were written in Pascal. The function definition must begin with the `pascal` keyword. Make sure you provide a return type. If your function needs to behave like a Pascal procedure, the return type is `void`.

For parameter declarations, follow the same rules as for calling Toolbox functions. Remember that non-`var` parameters are passed by value, not by reference. If the size of a non-`var` parameter is greater than 4 bytes, pass its address, but do not modify the parameter in your function.

For example, `ModalDialog()` lets you provide a `filterProc` to handle events in your dialog. This is how *Inside Macintosh* declares `ModalDialog()`:

```
PROCEDURE ModalDialog (filterProc: ProcPtr;
   VAR itemHit: INTEGER);
```

`ModalDialog()` expects the `filterProc` to have this declaration:

```
FUNCTION MyFilter (theDialog: DialogPtr;
   VAR theEvent: EventRecord;
   VAR itemHit: INTEGER) : BOOLEAN;
```

In THINK C and Symantec C++, the declaration for `MyFilter()` would look like this:

```
pascal Boolean MyFilter(DialogPtr theDialog
   DialogPtr theDialog,
   EventRecord *theEvent, short *itemHit)
```

The call to `ModalDialog()`, then, looks like this:

```
void F(void)
{
   extern pascal Boolean MyFilter();
   short theItem;

   ...
   ModalDialog(MyFilter, &theItem);
   ...
}
```

Keeping C and Pascal on speaking terms can be tricky, but THINK C tries to make it as simple as possible.

Calling Pascal routines indirectly

Only Macintosh Toolbox routines and functions that are declared `pascal` are called using Pascal conventions. If you have a pointer to a Pascal routine, you can call it indirectly through a pointer, the same way you would a routine in C, but the pointer must be of type pointer-to-pascal-function.

For example, this type definition defines `PFunc` as a pointer to a Pascal function that takes one `short` argument and returns `short`:

```
typedef pascal short (*PFunc)(short);
```

This function takes a Pascal routine as an argument and calls it indirectly:

```
short DoPascal(PFunc theFn, short i)
{

// theFn is a pointer to a pascal function
// that returns short

   i = (*theFn)(i);  // Call it indirectly

   i = theFn(i);     // This works too, but
                     // it's not obvious
                     // that it's called
                     // indirectly.
   return (i);
}
```

Now suppose you have a function declared `pascal` like this:

```
pascal short Add1(short v)
{
   return v + 1;
}
```

You could call the `DoPascal()` function defined above with it like this:

```
j = DoPascal(Add1, 10);
```

For more information about floating-point types in THINK C, see Chapter 29, "Language Reference." For more information about floating-point types in Symantec C++, see the Symantec C++ Compiler Guide*.*

Working with Floating-Point Formats

THINK C lets you use different floating-point formats, so you can take advantage of any special floating-point hardware or write libraries that will work regardless of the client project's settings. The main issues surrounding floating point and the Macintosh Toolbox involve SANE, the Standard Apple Numerics Environment, and the `extended` type.

Working with SANE

For more information on SANE, read the Apple Numerics Manual, Second Edition *(Addison-Wesley).*

To use SANE, add the SANE library to your project and include the file `SANE.h` in your source file. This library implements the SANE functions as well as some utility functions such as `x80tox96()`, `x96tox80()`, `str2dec()`, `cstr2dec()`, and `dec2str()`.

These eight functions are defined in both the SANE and ANSI libraries:

```
atan()     cos()      exp()      fabs()
log()      sin()      sqrt()     tan()
```

To use the SANE versions, include the file `SANE.h` before the file `math.h`. Similarly, to use the ANSI versions, include the file `math.h` before the file `SANE.h`.

The SANE library uses 80-bit `extended` values, so if your Compiler Settings are other than SANE-native (Generate 68881 Instructions off and Native Floating-Point on), be sure to use the `x80tox96()` and `x96tox80()` functions to convert between formats as in the example on the next page.

The extended type

Some Toolbox routines, and most SANE routines, use the type `extended` for floating-point numbers. In THINK C, the type `extended` is always an 80-bit SANE extended floating-point number, regardless of the Generate 68881 Instructions and Native Floating-Point settings. When you're using SANE-Native mode (Generate 68881 instructions off and Native Floating-Point on), the type `extended` is equivalent to `long double`, and you can use arithmetic and assignment operators on `extended` values.

When Generate 68881 Instructions is on, or when Native Floating-Point is off, the type `extended` is a `struct`. You can still use it to hold `extended` values, but you'll need to convert them to 96-bit values before you can use the arithmetic or assignment operators with them.

For example, the Toolbox routine `Frac2X` takes a `Fract` value and converts it to an `extended` value. The following THINK C function

converts a `Fract` value to a `long double` value, regardless of the compiler settings:

```
long double Frac2Dbl (Fract f)
{
   long double result;

#if !__option(mc68881) &&
     __option(native_fp)
   result = Frac2X(f);
#else
   {
      extended temp;

      temp = Frac2X(f);
      x80tox96(&temp, &result);
   }
#endif
   return (result);
}
```

For a description of the `__option` *directive, see "Accessing Option Settings in Your Code" in Chapter 15.*

To use the `x80tox96()` and `x96tox80()` functions, you need to add the SANE library to your project. These are the function prototypes for `x80tox96()` and `x96tox80()`:

```
void x80tox96(extended *x80,
   long double *x96);
void x96tox80(long double *x96,
   extended *x80);
```

The 96-bit value that `x80tox96()` stores in its second argument is in the THINK C universal floating-point format, so it's compatible with the MC68881.

A note on SANE implementations

The SANE library provided with THINK C is not identical to the SANE library provided by Apple with MPW C, though it has the same functionality.

MPW uses two SANE libraries—one for MC68881 code generation and one for straight MC680x0 code generation—because the size of an MPW `extended` changes if the compiler is generating code for the MC68881. The THINK C `extended` is always the same size, 80 bits, regardless of code generation options.

If you're curious, the sources for the THINK C SANE library are included in your THINK C package.